2. Desarrollo de Sistemas de Tiempo Real

Contenido

2.1 Introducción

Para abordar el tema del desarrollo de los Sistemas de Tiempo Real, es bueno pensar en el ciclo de vida del software y adaptarlo para este tipo de sistemas.

En el ciclo de vida de un sistema podemos distinguir las fases de análisis, especificación, diseño, codificación, verificación, instalación y mantenimiento.

Para los Sistemas de Tiempo Real son muy importantes las primeras fases del ciclo de vida, pues, un error en estas fases preliminares pueda hacer inabordable el proyecto. No hay que olvidar que las restricciones temporales son muy severas, fáciles de comprobar pero difíciles de asegurar su cumplimiento, sobre todo si el sistema contiene cierta complejidad. Por ello, es conveniente poder comprobar el comportamiento temporal en la fase más temprana posible.

El desarrollo de los distintos pasos en el ciclo de vida, una tras otro, tiende a reducir el tiempo dedicado a la fase de especificación, con sus consecuencias negativas.

Aunque la fase de especificaciones es suficientemente importante para que sea larga, sin embargo, la experiencia aconseja no retrasar demasiado el comienzo de la fase de codificación, comenzándola incluso antes de que la fase de especificación haya concluido o madurado [Levi90].

Un buen método es comenzar la parte de la codificación construyendo unos prototipos que puedan mostrar, cuanto antes, un funcionamiento parcial del sistema. Este prototipo servirá para validar los criterios asumidos en la parte de la especificación y, a su vez, mientras se desarrolla, puede ayudar a validar o a rechazar líneas tomadas en la especificación.

2.2 Especificaciones

El desarrollo de unas especificaciones de requerimientos completas y consistentes se extiende a lo largo del ciclo de vida del sistema.

En la elaboración de las especificaciones de un sistema es importante la notación empleada. Se pueden distinguir tres niveles de notación:

- **Informal**: Usualmente hace uso del lenguaje natural y varias formas de diagramas, generalmente imprecisos. Tiene la ventaja que lo puede entender cualquiera que conozca el idioma empleado.

- **Estructurada**: A menudo usa una representación gráfica pero, al contrario que en los diagramas de la notación informal, estos están bien definidos. Se construyen a partir de un pequeño número de componentes predefinidos que se interconectan de una manera controlada. Aunque es una notación rigurosa, no puede ser analizado y manipulado de forma automática.

- **Formal**: Está basada en un lenguaje matemático y, por tanto, puede manipularse utilizando propiedades y métodos matemáticos. Tiene la ventaja que se pueden hacer una descripción muy precisa y, además, se puede hacer un estudio de las propiedades que poseerá nuestro sistema, como por ejemplo, que el diseño satisface la especificación de requerimientos. La desventaja del método formal es que no se entiende fácilmente si no se tiene un dominio del lenguaje matemático utilizado en la notación.

La alta seguridad que requieren los sistemas de tiempo real está provocando que, cada vez más, se estén utilizando métodos formales. Con estos se utilizan también técnicas rigurosas de verificación, pero pocos ingenieros de software poseen los conocimientos matemáticos necesarios para explotar totalmente las posibilidades de verificación.

En la práctica la notación formal es útil cuando se dispone de la herramienta que la maneje y realice el trabajo tedioso.

2.3 Diseño

El diseño de un sistema de tiempo real complicado no se puede resolver como si se tratara de un ejercicio. A menudo ni siquiera es resuelto por una solo persona. La solución se debe estructurar de alguna manera.

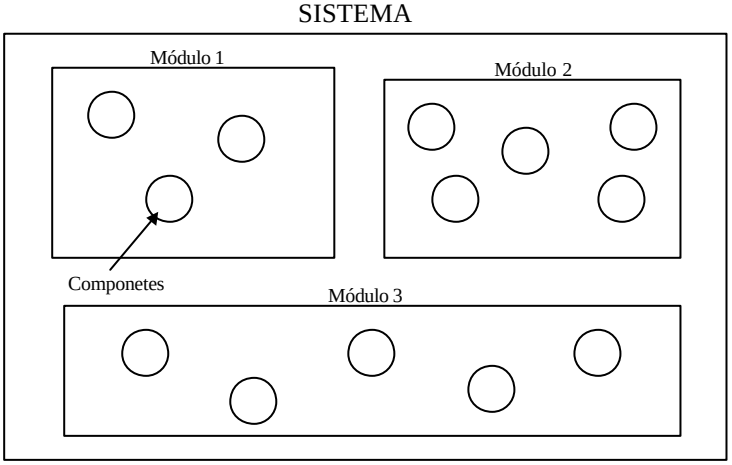
El diseño se debe descomponer en dos aspectos, el diseño estructural y el diseño detallado. Si un sistema es medianamente complicado resulta prácticamente inabordable si no se descompone es subsistemas más sencillos relacionados entre sí.

El diseño estructural consiste en la forma en que se descompone el sistema y como se relacionan los distintos subsistemas.

El diseño detallado consiste en el desarrollo de cada uno de los subsistemas obtenidos en la fase anterior. Pude realizarse en varios niveles, es decir, un subsistema se puede descomponer en varios subsistemas más sencillos.

Cuanto más detallado sea el diseño y más independientes resulten los subsistemas, más sencillas resultarán las fases siguientes del ciclo de vida del sistema.

Cada. subsistema lo denominaremos módulo, y cada módulo estará formado por un conjunto de componentes:



Descomposición de un sistema

Para manejar el desarrollo de sistemas de tiempo real complejos, a menudo se utilizan dos técnicas complementarias: La descomposición y la abstracción.

La **descomposición** consiste en dividir un sistema complejo en módulos cada vez más pequeños y aislados, de forma que puedan ser abordados por un único ingeniero o un grupo reducido.

La **abstracción** permite posponer para más adelante las consideraciones de detalle, especialmente las correspondientes a la implementación. Esto permite una visión simplificada del sistema y de los objetos que contiene, los cuales contienen las propiedades y características esenciales.

En la descomposición es importante que cada módulo forme una unidad independiente lo más aislada posible del resto de componentes (encapsulación). Existen dos criterios para obtener una buena encapsulación: la cohesión y el acoplamiento.

La **cohesión** hace referencia a la conexión interna de los componentes de un módulo. Existen seis valores (según Allworth y Zobel - 1987) en la medida de la cohesión, que va desde muy poca hasta mucha:

- **casual**: los elementos de un módulo no están ligados entre sí más que de forma superficial, como por ejemplo que se han escrito en el mismo mes.
- **lógica**: los elementos de un módulo están relacionados entré si en relación al sistema en general, y no en torno al software del proyecto actual, como por ejemplo los drivers de entrada / salida de los periféricos.
- **temporal**: los elementos del módulo se ejecutan en instantes similares, como por ejemplo las rutinas de puesta en marcha.
- **procesal**: los elementos del módulos se utilizan en la misma sección del programa, como por ejemplo los componentes de interface de usuario.
- **comunicacional**: los elementos del módulos funcionan sobre la misma estructura de datos, como por ejemplo los algoritmos utilizados para analizar una señal de entrada.
- **funcional**: los elementos del módulo se utilizan conjuntamente para realizar una determinada función del sistema, como por ejemplo el servir un sistema de ficheros distribuido.

El **acoplamiento**, por el contrario, es una medida de la interdependencia entre los módulos de un programa. Si dos módulos se pasan información de control entre ellos, se dice que tienen un grado bajo de acoplamiento. Y al contrario, el acoplamiento es alto si únicamente se pasan datos. Otra forma de saber si el acoplamiento de un módulo con el resto es bajo es medir la facilidad de eliminar o reemplazar este módulo por otro alternativo.

En todos los métodos de diseño, una buena descomposición es aquella que tiene una fuerte cohesión en todos sus módulos y un grado de acoplamiento bajo entre ellos. Este principio es válido tanto en la programación secuencial como en la concurrente.

Existen muchas metodologías de diseño que han sido adaptadas para los sistemas de tiempo real. Estos son interesantes en la medida que van acompañados de herramientas de software que facilite su utilización. Algunos de ellos son:

HOOD (Hierarchical Object-Oriented Design)
JSD (Jackson System Development)

DARTS (Design of Ada Real - Time Systems)
SA/SD RT (Structured Analysis / Structured Design for Real - Time)
PAMELA (Process Abstraction Method for Embedded Large Applications)
Otros (Mascot3, Booch, Buhr, Redes de Petri, etc)

De estos vamos a estudiar solamente una versión del HOOD adaptada para sistemas de tiempo real duros llamado HRT-HOOD.

2.3.1 HRT-HOOD

Esta metodología tiene en cuenta las necesidades de los sistemas de tiempo real. En ésta el proceso de diseño se ve como una progresión creciente de compromisos específicos.

Los compromisos definen las propiedades que se deben ir cumpliendo en el diseño del sistema y que el diseñador no puede modificar en niveles de mayor detalle.

Aquellos aspectos del diseño que no suponen un compromiso en un determinado nivel se deben asumir como obligaciones en los niveles inferiores (mayor detalle) del sistema.

Las obligaciones en el nivel superior corresponden a las propiedades que aparecen en las especificaciones.

Los compromisos son las concreciones que se hacen en el diseño que hacen que se cumplan las propiedades correspondientes a las obligaciones.

El proceso de refinamiento del diseño consiste en ir transformando las obligaciones en compromisos. Este proceso está sujeto a restricciones impuestas principalmente por el entorno de ejecución. El entorno de ejecución es un conjunto de componentes software y hardware sobre los que se construye el sistema.

Como ejemplo podría servir la construcción de un chalet. Esta sería la primera obligación que se traduce en los siguientes compromisos (obtención de permisos, búsqueda del préstamo, diseño de chalet y su construcción).

Cada uno de estos compromisos se pueden describir con una serie de obligaciones que, a su vez, se podrán concretar en compromisos cada vez más detallados y sencillos.

En HRT-HOOD se definen dos tipos de actividades en el diseño de la arquitectura:

- Diseño de la arquitectura lógica.

- Diseño de la arquitectura física.

La arquitectura lógica se ocupa de los compromisos que se pueden hacer con independencia de las restricciones impuestas por el entorno de ejecución. Supone principalmente los requerimientos funcionales.

La arquitectura física tiene en cuenta los requerimientos funcionales y, además, las restricciones del entorno. Abarca además los requerimientos no funcionales. En la arquitectura física es donde se tienen en cuenta las restricciones temporales.

Aunque la arquitectura física es un refinamiento de la lógica, ambas se desarrollan y modifican a la vez. Las técnicas de análisis concretadas en la arquitectura física se deben aplicar lo antes posible. Estas afectaran a los recursos iniciales previstos, que deberán revisarse y modificarse según se vaya refinando la arquitectura lógica (sin esperar a concluir ésta para empezar con la arquitectura física, de lo contrario se podría invalidar demasiado tarde los recursos asumidos en la primera).

2.3.2 Ejemplo de diseño. Drenaje de una mina

Para ilustrar esta metodología, vamos a aplicarla sobre un ejemplo clásico en la literatura. Un sistema software encargado de controlar el drenaje de una mina en un entorno mínimo.

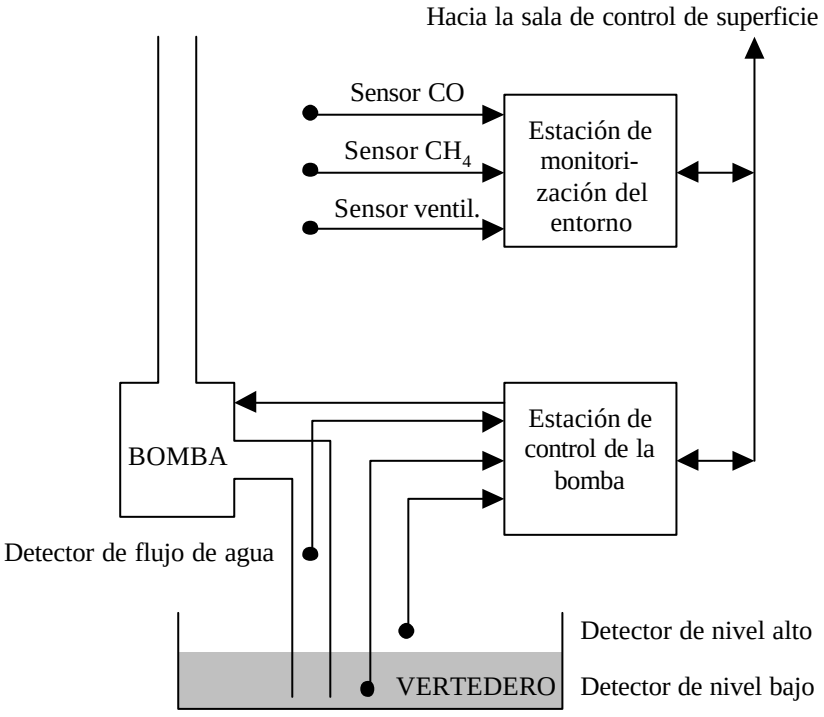
El sistema se utiliza para extraer agua de una mina, que está depositada en un vertedero, desde el fondo del pozo hasta la superficie.

El drenaje se debe realizar bajo unas condiciones mínimas de seguridad. El requerimiento de seguridad mayor es que la bomba no debe funcionar cuando el nivel de metano exceda un umbral de seguridad, bajo riesgo de una explosión.

Además del sensor de metano en la mina, se dispone de sensores para medir el nivel de agua en el vertedero (nivel máximo y nivel mínimo), flujo de agua en la tubería de extracción, medidor de ventilación de aire y medidor de nivel de monóxido de carbono.

Existe además una estación de control desde la que se podrá controlar el funcionamiento de la bomba y monitorizar el estado y las alarmas del sistema.

También existirá un registro en base de datos de todo lo que ocurre en el sistema.



Sistema de control de drenaje de una mina

2.3.2.1 Requerimientos funcionales

Los requerimientos funcionales del sistema se pueden dividir en cuatro componentes: la operación de la bomba, la monitorización del entorno, la interacción del operador y la monitorización del sistema.

Operación de la bomba.

Se debe monitorizar el nivel de agua en el vertedero. Cuando el nivel supere el nivel máximo o lo requiera el operador, se debe conectar la bomba, y si el nivel desciendo del nivel mínimo o lo requiere el operador, se debe detener la bomba.

Además se debe monitorizar el flujo de agua por la tubería, independientemente del estado de la bomba, generando una alarma si el flujo no corresponde a la operación de la bomba.

La bomba no se debe permitir que funcione cuando el nivel de metano exceda el máximo permitido.

Monitorización del entorno.

Se deben monitorizar las siguientes variables del entorno:

- Nivel máximo de metano en la mina.
- Nivel de monóxido de carbono en la mina.
- Funcionamiento correcto de la ventilación

Sí el nivel de metano supera el máximo permitido se debe detener el funcionamiento de la bomba.

Si se supera el nivel de monóxido de carbono o la ventilación no funciona correctamente se deben generar las correspondientes alarmas.

Interacción del operador.

El sistema es controlado desde la superficie por medio de la consola del operador. Por medio de ella se informa al operador del estado de las alarmas y le permite activar o desactivar la bomba.

Monitorización del sistema.

Todos los eventos producidos en el sistema se registran en una base de datos y se pueden consultar bajo demanda.

2.3.2.2 Requerimientos no funcionales

Los requerimientos no funcionales se pueden dividir en tres componentes:

- Comportamiento temporal (timing)
- Confianza (dependability)
- Seguridad (security)

En nuestro caso solo estudiaremos el comportamiento temporal, y éste ya incluirá los otros dos componentes.

Los requerimientos temporales en nuestro sistema son:

Periodos de monitorización.

Los periodos máximos para la lectura de sensores del entorno puede que estén determinados por la legislación. En nuestro ejemplo supondremos que el periodo máximo es el mismo para todos los sensores y será de 100 ms.

En el caso de los detectores de CH_4 y de CO existe una restricción y es que por las características del sensor, es necesario cierto tiempo para obtener una medida. En nuestro caso el tiempo de desfase entre que se solicita una medida y se obtiene el resultado es de 40 ms. para ambos sensores. Esto nos da un tiempo máximo de tratamiento de la medida de 60 ms. para no solaparnos con el siguiente ciclo.

El detector de flujo de agua tiene otra peculiaridad y es que existe cierta inercia desde que se detiene la bomba (se pone en marcha) y el agua deja de (empieza a) circular.

Debido al tiempo de retraso en el flujo del agua, la medida se realiza cada segundo y se tomarán dos muestras consecutivas para determinar el estado del flujo. Para asegurarse que el tiempo entre muestras es exactamente de 1 segundo, se considerará un tiempo de respuesta bastante exigente para la medida con plazo de 40 ms. lo que dará una margen entre medidas que irá desde 960 ms. a 1040 ms.

En cuanto a los detectores de nivel del agua, suponemos que son manejados por interrupción, lo que supone que se generan eventos de forma esporádica. Las condiciones físicas de nuestro sistema (caudal máximo de entrada y flujo de extracción de la bomba) nos permiten asegurar que como máximo se producirán dos interrupciones consecutivas cada 6 seg. Como plazo para el tratamiento de estos eventos le asignamos 200 ms.

Tiempo límite de parada.

Para evitar explosiones se debe fijar un tiempo máximo de reacción entre que se supera el umbral permitido de metano y se detiene la bomba.

El tiempo límite de parada vendrá dado por la siguiente desigualdad:

$$R(T + D) < M$$

Donde:

- R: Velocidad máxima de acumulación del metano
- T: Periodo de muestreo
- D: Tiempo límite de parada
- M: Margen de seguridad ajustado en el sensor

Podemos asumir que esta desigualdad se cumple en nuestro caso con D = 30 ms., con un periodo de muestreo de T = 80 ms.

Tiempo límite en la información al operador.

El operador debe ser informado antes de 1 seg. de la detección de un nivel crítico de metano o de monóxido de carbono, antes de 2 seg. de un valor crítico en la ventilación y de 3 seg. en el fallo en la operación de la bomba (fallo en el flujo de agua).

2.3.2.3 Diseño con la metodología HRT-HOOD

HRT-HOOD facilita el diseño de la arquitectura lógica de un sistema ofreciendo diferentes tipos de objetos. Estos son:

- **Pasivos**: Objetos reentrantes que no tiene control sobre cuando son invocadas las operaciones que ejecutan, y no invocan de forma espontánea operaciones en otros objetos.

- **Activos**: Objetos que pueden controlar cuando se ejecutan las invocaciones de sus operaciones, y pueden espontáneamente invocar operaciones de otros objetos. Los objetos activos son la clase más general de objetos y no hay restricciones impuestas sobre ellos.

- **Protegidos**: Objetos que pueden controlar cuando se ejecutan las invocaciones de sus operaciones, pero no pueden invocar operaciones de otros objetos de forma espontánea; en general los objetos protegidos no pueden tener restricciones de sincronización arbitrarias y deben ser analizados los tiempos de bloqueos que imponen a sus llamantes.

- **Cíclicos**: Objetos que representan actividades periódicas; pueden invocar espontáneamente operaciones de otros objetos y tienen sólo interfaces muy restrictivos.

- **Esporádicos**: objetos que representan actividades esporádicas; los objetos esporádicos pueden invocar operaciones de otros objetos de forma espontánea; cada uno tiene una operación simple que es llamada cuando es invocado el objeto esporádico.

Si llamamos P1 a la propiedad de poder controlar cuando se ejecutan las operaciones invocadas desde otros objetos, y P2 la propiedad de poder invocar operaciones de otros objetos de forma espontánea, tendremos:

TIPO	**P1**	**P2**	**COMENTARIO**
Pasivo	No	No	Máxima restricción
Activo	Sí	Sí	Sin restricciones
Protegido	Sí	No	Bloqueos controlados
Cíclico	Sí	Sí	Periódicas. Interfaz sencilla
Esporádico	No	Sí	Una única operación

Un programa de tiempo real duro diseñado con HRT-HOOD debe contener en el nivel final (esto es, después de la descomposición completa) solo objetos cíclicos, esporádicos, protegidos y pasivos. Los objetos activos, debido a que al carecer de restricciones, no pueden ser analizados totalmente, sólo se deben permitir para actividades de fondo (background). Los objetos activos se pueden utilizar durante la

descomposición, pero deben ser transformados en uno de los otros tipos de objetos antes de alcanzar el nivel final.

2.3.2.4 El diseño de la arquitectura lógica

Cada uno de los componentes de la descomposición funcional se debe modelar como un objeto que ofrece una serie de operaciones (compromisos), que realiza una serie de acciones y que puede llamar a operaciones (obligaciones).de otros objetos

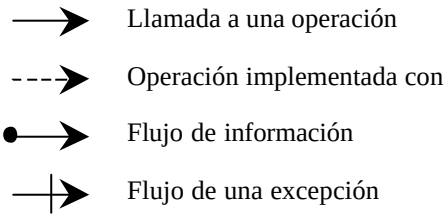
En cada llamada a una operación puede haber un flujo de información que puede recibir el invocador, el llamado o ambos.

Un objeto también puede lanzar una excepción sobre otro objeto.

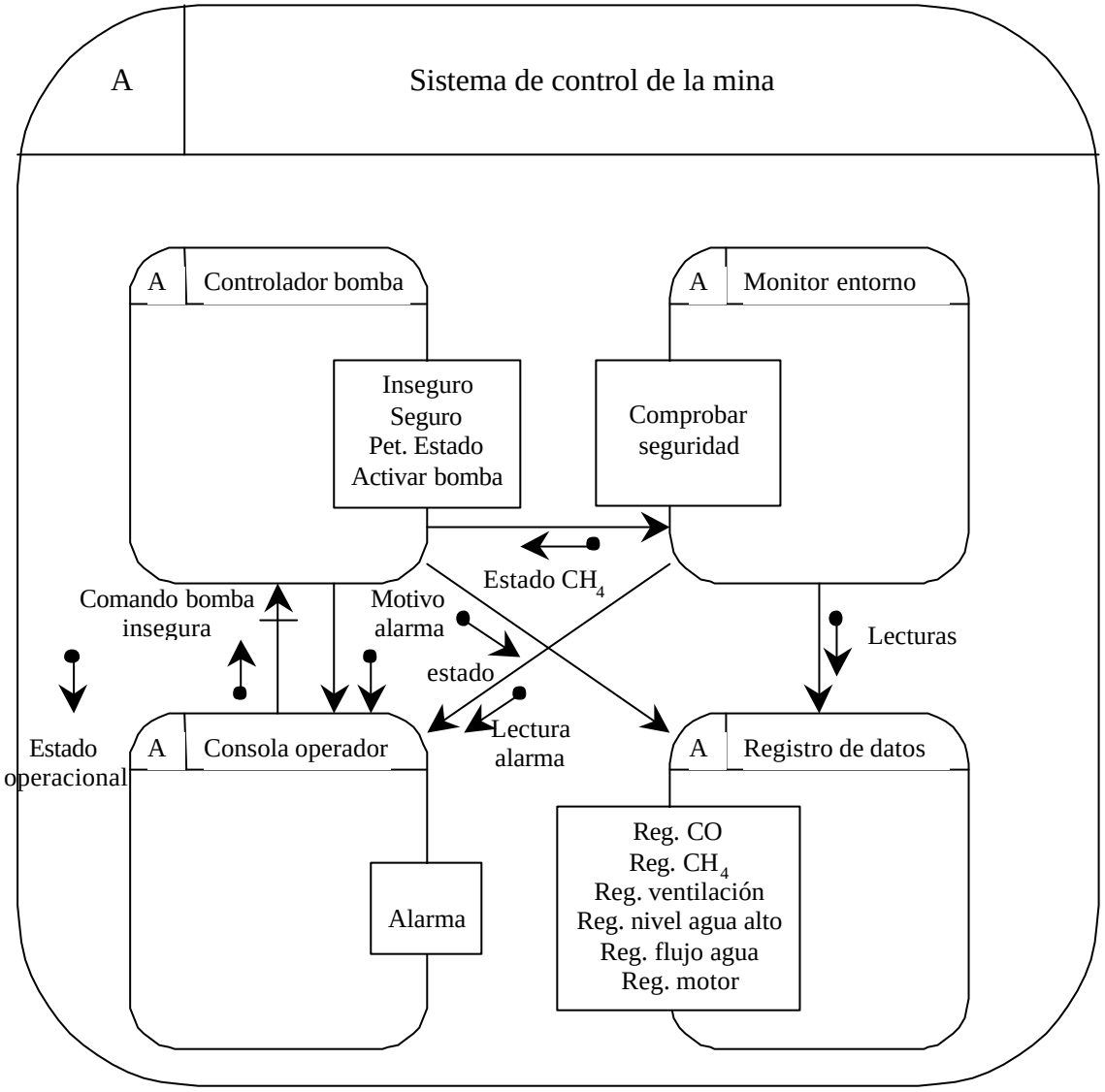
Partiendo de la arquitectura lógica, los subsistemas del primer nivel serían:

- Control de la bomba
- Monitor del entorno
- Consola del operador
- Registro de datos

La interacción entre los distintos subsistemas se representa en el diagrama, utilizando unos símbolos predefinidos formados por flechas que indican:



Con esto, la representación del primer nivel de descomposición de nuestro sistema, sería el siguiente:

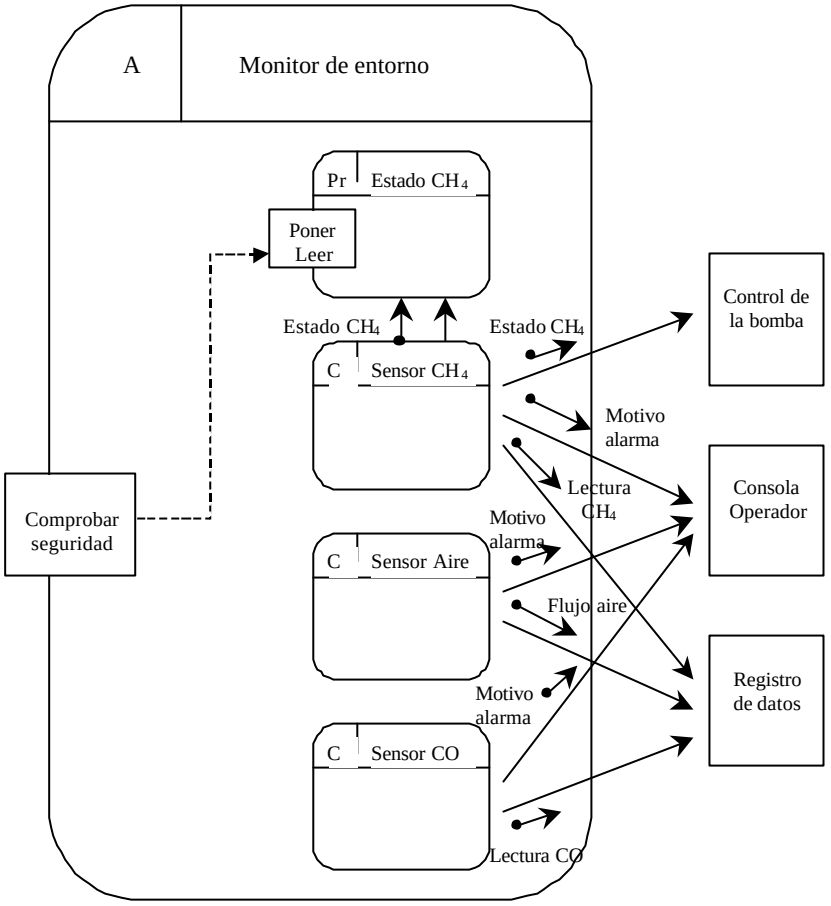


En la parte superior se indica el nombre del módulo y en la esquina izquierda de cada objeto del tipo que es (A = Activo, P = Pasivo, Pr = Protegido, C = Cíclico y S = Esporádico).

En un recuadro al margen de la caja del subsistema se indican las operaciones que ofrece este subsistema para el resto.

Cada objeto puede tener varios niveles de refinamiento hasta llegar a la máxima descomposición formada por los objetos permitidos por la metodología. No hay que olvidar que hay que transformar los objetos activos en otro tipo de objetos.

Por ejemplo, el monitor de entorno tendría la siguiente descomposición:



En el que ya no queda ningún objeto activo

2.3.2.5 El diseño de la arquitectura física

HRT-HOOD soporta el diseño de la arquitectura física a partir de:

- La definición de atributos temporales asociados a cada objeto.
- Ofreciendo un marco desde el cual se puede definir una aproximación de la planificabilidad y el análisis de los objetos finales que tienen lugar.
- Ofreciendo una abstracción con la cual el diseñador puede expresar el manejo de los errores temporales.

En la práctica esto supone definir para cada objeto periódico o esporádico su periodo, su tiempo límite y su prioridad. Estos datos se pueden extraer de los requerimientos no funcionales.

Asignado un tiempo de ejecución en el peor de los casos (WCET) a cada objeto, que puede ser calculado o medido experimentalmente, se puede realizar una análisis de planificabilidad y comprobar si el diseño es válido o requiere ser modificado.

En nuestro ejemplo, una vez finalizada la descomposición, tendríamos:

Objeto	Tipo	Periodo	Límite	Prioridad	WCET
Sensor CH_4	Cíclico	80	30	10	12
Sensor CO	Cíclico	100	60	8	10
Sensor flujo aire	Cíclico	100	100	7	10
Sensor flujo agua	Cíclico	1000	40	9	10
Manejador nivel agua	Esporádico	6000	200	6	20
Motor	Protegido			10	
Controlador nivel agua	Protegido			11	
Estado CH_4	Protegido			10	
Consola del operador	Protegido			10	
Registro de datos	Protegido			10	

Como se puede apreciar, en el nivel último de la descomposición no queda ningún objeto activo.

2.3.2.6 Codificación

HRT-HOOD soporta una traducción sistemática de sus objetos y su interacción sobre el lenguaje Ada.

Para cada objeto terminal se generan dos paquetes. El primero contiene simplemente una colección de tipos de datos y variables definiendo los atributos de tiempo real del objeto; el segundo contiene el código propio del objeto.

2.4 Codificación.

2.4.1 Lenguajes de programación

Podemos identificar tres clases de lenguajes de programación que se utilizan, o se han utilizado, en el desarrollo de los sistemas de tiempo real. Estos son: lenguajes ensambladores, lenguajes secuenciales y lenguajes concurrentes.

Lenguajes ensamblador

El lenguaje ensamblador se utilizó inicialmente en la programación de los sistemas empotrados. Esto fue así porque muchos procesadores no soportaban los lenguajes de programación de alto nivel y los lenguajes ensambladores eran los únicos que ofrecían la eficiencia que requerían los sistemas de tiempo real.

El problema principal es que son lenguajes orientados a la máquina, y no al problema. El programador debe hacer la traducción entre las elementos que intervienen en su sistema y los que es capaz de representar el lenguaje, con el consiguiente costo de desarrollo y mantenimiento.

Otro problema es la escasa o nula portabilidad, que hace un problema resuelto en una máquina no sea trasladable a otra.

Se siguen usando en microcontroladores (sistemas con poca memoria y baja velocidad de cálculo), pero cada vez menos debido al abaratamiento de los sistemas más potentes.

Lenguajes secuenciales

A medida que los ordenadores se hicieron más potentes, los lenguajes de programación evolucionaron, así como la tecnología de los compiladores. Desaparecieron entonces los inconvenientes que tenían frente al lenguaje ensamblador y se difundió su uso. Para hacer frente a las deficiencias que aparecían en los primeros lenguajes (como el FORTRAN) se desarrollaron nuevos lenguajes especializados en los sistemas empotrados. De estos nuevos lenguajes los más recientes son el C y el C++.

Todos estos lenguajes tienen en común que son secuenciales. También suelen presentar deficiencias frente a las necesidades de la programación en tiempo real. Para resolver esto a menudo es necesario incluir partes en ensamblador y recurrir a un sistema operativo que de soporte a la concurrencia. La desventaja que esto presenta es que el lenguaje queda incompleto para soportar totalmente la programación de tiempo real, presentando los problemas de portabilidad, bien debidos a la dependencia de la máquina o del sistema operativo utilizado.

Lenguajes concurrentes

Pese al incremento del uso del software en el desarrollo de sistemas y la evolución de los lenguajes de programación, la producción de software se hizo cada vez más complicada y frustraste a partir de los años 70, a medida que los sistemas basados en ordenador se hacían cada vez más grandes y sofisticados. Ocurrió que a medida que los precios del hardware disminuían, el precio del software se disparaba.

A la aparición de estos problemas se le denomina la *crisis del software*. Hay varios síntomas que permiten reconocer la aparición de esta crisis:

- **Falta de respuesta**. A menudo la producción de los sistemas que han sido automatizados no responden a las necesidades del usuario.
- **Fiabilidad**. El software no es fiable y, a menudo, falla a la hora de cumplir totalmente las especificaciones
- **Coste**. El coste del software es raramente predecible
- **Modificabilidad**. El mantenimiento del software es complejo, costoso y fácilmente introduce nuevos errores
- **Retraso**. Los proyectos de software a menudo se terminan con retraso.
- **Transportabilidad**. El software desarrollado en un sistema raramente funciona tal cual en otro.
- **Eficiencia**. El coste en el desarrollo del software no consigue hacer un uso óptimo de los recursos que utiliza.

Esta realidad obligó a una evolución en las metodologías de programación y a una mejora de los lenguajes que resolvieran o minimizaran los problemas anteriores.

De hecho, el lenguaje Ada nació como fruto de un estudio realizado por el Departamento de Defensa de los EEUU para resolver el enorme problema que les había aparecido en su producción de software.

No sólo apareció el Ada y sino que otros lenguajes también evolucionaron. Por ejemplo de Módula apareció Modula-2 como un lenguaje más de propósito general, del C apareció el C++

soportando directamente la programación orientada a objetos, hasta incluso apareció una versión de BASIC para tiempo real que, aunque carecía de muchas características de los lenguajes de tiempo real como los tipos definidos por el usuario, ofrecía facilidades de tiempo real como podía ser la concurrencia (Ada y Módula son lenguajes concurrentes).

También aparecieron otros lenguajes como el PERL, utilizado en Alemania para aplicaciones de control de procesos, Mesa, utilizado por Xerox (Xerox Corporation, 1985) en sus equipos de automatización de oficina y CHILL (CCITT, 1980), que desarrolló CCITT para la programación de sus aplicaciones de telecomunicaciones.

2.4.2 Criterios generales en el diseño de lenguajes

Aunque un lenguaje para tiempo real debe satisfacer, sobre todo, las necesidades de la programación de los sistemas empotrados, su uso raramente se limita a esta área. Muchos lenguajes de programación de tiempo real también se utilizan en la implementación de sistemas de propósito general como pueden ser la creación de compiladores o de sistemas operativos.

Young (1982) plantea seis criterios que se deben cumplir en los lenguajes de programación de tiempo real. Estos son:

Seguridad

Consiste en la capacidad para detectar errores de forma automática, tanto durante la compilación como en el tiempo de ejecución a través del 'run-time support' del lenguaje.

Naturalmente existe un límite en esta capacidad, como puede ser detectar errores en la lógica del programa producidos por el programador. De todas formas, un lenguaje seguro debe ser estructurado y facilitar la localización de errores que no cumplan con su estructura.

La seguridad aporta los siguientes beneficios:

- La detección de errores en fases tempranas del desarrollo de un programa, generalmente reduce considerablemente su coste (por ejemplo la comprobación de la coherencia de tipos, de la que carece las primeras versiones de C).

- La reducción de la sobrecarga. Las comprobaciones que se hagan en tiempo de compilación permiten reducir las comprobaciones en tiempos de ejecución y, por tanto, el programa será más rápido.

Legibilidad

La legibilidad de un lenguaje depende de un gran número de factores, incluyendo la elección de las palabras reservadas, la posibilidad de crear nuevos tipos y la capacidad para realizar programas modulares.

Un programa será más legible cuanto mejor se entienda su código, sin necesidad de comentarios detallados ni diagramas que expliquen su funcionamiento.

Las ventajas de una buena legibilidad incluye:

- Reduce el costo de la documentación
- Aumenta la seguridad
- Disminuye los costos de mantenimiento

Flexibilidad

Un lenguaje debe ser suficientemente flexible para permitir al programador expresar todas las operaciones que necesita con claridad y de forma coherente. En el caso contrario, como ocurría en los lenguajes primitivos, el programador debía acudir a llamadas al sistema operativo para realizar ciertos comandos, o incluir pedazos de código máquina para poder realizar ciertas operaciones.

Simplicidad

La simplicidad se entiende como lo opuesto a la complejidad, en el sentido que el lenguaje no esté formado por un gran número de palabras y un gran número de construcciones y posibilidades para hacer una misma cosa.

En los lenguajes de programación la simplicidad aporta las siguientes ventajas:

- minimiza el esfuerzo necesario para producir compiladores
- reduce el coste asociado al aprendizaje de los programadores
- disminuye la posibilidad de cometer errores debido a una incorrecta interpretación de las propiedades del lenguaje

La flexibilidad y la simplicidad están asociadas con el **poder de expresión** y la **facilidad de utilización** del lenguaje respectivamente.

La legibilidad y la simplicidad pueden ser criterios opuestas (algunas operaciones complejas expresadas en un lenguaje excesivamente sencillo pueden resultar ilegibles). Se debe buscar un compromiso entre estos dos criterios.

Portabilidad

Un programa, hasta cierto punto, debería ser independiente del hardware en el que se ejecuta. En los sistemas de tiempo real esto es difícil de alcanzar pues, normalmente, una parte sustancial del programa manipula los recursos que ofrece el hardware. No obstante, el lenguaje debe ser capaz de aislar la parte *dependiente* de la máquina de la parte *independiente* de la máquina.

Eficiencia

En los sistemas de tiempo real se debe garantizar el tiempo de respuesta, por la tanto, el lenguaje debe producir código eficiente (código optimizado). De lo contrario se estará perdiendo capacidad de control al aumentar el tiempo de cómputo de los algoritmos empleados. Un ejemplo es la implementación de la estructura switch, por una cadena de if's o por una tabla de saltos.

Por lo general, los lenguajes interpretados son menos eficientes que los lenguajes compilados y, por tanto, menos adecuados para ser usados en los sistemas de tiempo real.

Teniendo en cuenta el comportamiento predecible que deben ofrecen los sistemas de tiempo real, se deben evitar sobre todo los mecanismos que produzcan un tiempo de sobrecarga indeterminado

Obviamente, la necesidad de eficiencia se debe balancear con la seguridad, la flexibilidad y la legibilidad que repercuten negativamente en la eficiencia.

2.4.3 Ada

El lenguaje Ada es un buen ejemplo de lenguaje adecuado para la programación de sistemas en tiempo real. Es el único lenguaje que se ha creado para cumplir unas especificaciones, las cuales estaban muy ligadas a las necesidades de la programación de los sistemas empotrados.

Además de respetar de una forma aceptable los criterios que deben cumplir los lenguajes de tiempo real, soporta la programación concurrente que es deseable para la programación de la mayor parte de los sistemas de tiempo real.

2.5 Verificación

La fiabilidad requerida en la mayoría de los sistemas de tiempo real, sobre todo en los sistemas empotrados, hace necesaria un proceso de pruebas y verificación muy estricto.

En otro tipo de aplicaciones es admisible que el propio usuario realice las pruebas y detecte los errores que se puedan haber cometido. Esto se suele hacer, por ejemplo, en aplicaciones de gestión en las que, después de una primera instalación el usuario, a medida que las utiliza, va reportando los errores que encuentra al suministrador de la aplicación durante el periodo de garantía, y este los va resolviendo según aparecen. Estamos acostumbrados también a que muchos programas comerciales, y hasta sistemas operativos, aparecen inicialmente con una versión beta, que los futuros usuarios van utilizando y encontrando errores. O hasta es común que en las primeras versiones oficiales se encuentren errores.

En algunos productos comerciales que no son de tiempo real se intenta encontrar un compromiso entre el coste de desarrollo y el número de fallos del producto. Esto no es posible es sistemas de tiempo real empotrado, pues, la aparición de un error, no sólo podría llevar con sigo que el sistema dejara de funcionar, sino que podría causar la destrucción del sistema que controla. En estos casos no hay más remedio que coordinar una buena verificación con la tolerancia a fallos.

2.5.1 Localización de errores

La verificación no se limita solamente al producto final. La descomposición que se ha incorporado en el diseño ofrece una arquitectura adecuada para realizar una verificación parcial de cada uno de sus módulos. Hay que tener en cuenta que siempre es preferible localizar los errores de forma aislada en un módulo del sistema que no en el producto final, pues el tiempo necesario para detectarlo y corregirlo es mucho menor. Hay que tener en cuenta que la existencia de un error en una parte del programa puede manifestarse en otras, y hasta en un tiempo posterior, siendo muchas veces difícil localizar la causa de un problema cuando se observa en el producto final.

La dificultad que aparece en los programas concurrentes de tiempo real es que la mayoría de los errores intratables que aparecen son el resultado de una sutil interacción entre los procesos. A menudo los errores son también dependientes del tiempo y se manifiestan sólo en raras ocasiones. (Por ejemplo, un puntero sin inicializar).

Otro elemento que añade dificultad en la localización de errores es que estos también pueden aparecer como fruto de la interacción del sistema de control con el entorno que controla, y no siempre es fácil predecir todos los comportamientos posibles del sistema. Se debe prestar una atención especial a los caminos de recuperación de errores e investigar el efecto de varios fallos simultáneos.

2.5.2 Simuladores

Como ayuda en la actividad de verificación en sistemas complicados, se debe acudir en la fase final a pruebas reales. En la mayoría de los casos, en los que disponer del entorno real en le laboratorio es imposible o muy costoso, no hay más remedio que acudir a los simuladores.

Un simulador es un programa que imita las acciones del sistema en el cual se empotra el software de tiempo real. Simula la generación de interrupciones y realiza las operaciones de Entrada / Salida que esperan los programas de tiempo real, intentando imitar la reacción del entorno frente a los estímulos que proporcione nuestro sistema.

Utilizando un simulador se pueden crear tanto comportamientos normales y anormales del entorno. Pero lo más interesante es que se pueden reproducir situaciones a voluntad tantas veces se quiera, facilitando la reproducción de errores (cuando se produce un error, sí se es capaz de reproducir, se tiene adelantado el 50% del camino para resolverlo).

En ocasiones la construcción de simuladores es inevitable, pues no existe otra forma de probar el producto final, bien por las características del entorno (por ejemplo, la vasija de un reactor nuclear), o bien porque el sistema a controlar y el sistema de control son proyectos paralelos.

Los simuladores pueden ser programas que se ejecuten concurrentemente en el sistema de control, pero esto, aunque permite verificar el comportamiento lógico de los programas, presenta dificultades para verificar el comportamiento temporal debido a la interferencia que se produce al utilizar la misma CPU.

Para facilitar la verificación, interesa que el simulador permita probar los módulos usados en el diseño de forma independiente.

Aunque resulte más costoso, en ocasiones no hay más remedio que utilizar un sistema multiprocesador, o hardware independiente, para construir el simulador. Este simulador es en sí otro sistema de tiempo real y debe ser a su vez completamente verificado, si bien no precisa ser tan fiable y son tolerables fallos ocasionales.

La construcción de simuladores no es algo trivial, sobre todo en sistemas complicados, resultando una parte importante en el coste total del proyecto, incluyendo en ocasiones hardware específico. Por ejemplo, en la lanzadera espacial de la NASA, el coste de los simuladores superaba el coste del software de control de tiempo real, pero este dinero se consideró bien gastado cuando se detectaron varios fallos tras varias horas de simulación de vuelo.

2.5.3 Herramientas

Otro tema importante son las herramientas utilizadas para la verificación. No siempre son válidas las herramientas comerciales para monitorizar el comportamiento de nuestro sistema, o tienen un coste excesivo para el proyecto.

Otras veces la sobrecarga que introduciría una auto monitorización del software la hace no viable. En tal caso es necesario desarrollar herramientas que nos permitan "ver" como evoluciona nuestro sistema o como se comporta, tanto lógica como temporalmente. Lo ideal es integrar la monitorización en el simulador pero, si esto no es posible, habrá que desarrollar el equipo de monitorización adecuado.

2.6 Prototipos

Con la finalidad de validar en la fase más temprana posible del desarrollo de un sistema las especificaciones es conveniente la construcción de un prototipo.

Si las especificaciones no se validan hasta las últimas fases del proyecto (fase de verificación o instalación), se corre el riesgo de tener que rehacer gran parte del trabajo, o haber hecho hincapié en las partes que no son las más importantes.

El principal objetivo de un prototipo es ayudar a asegurarse que las necesidades y expectativas que el cliente tiene de nuestro desarrollo están perfectamente plasmadas en las especificaciones. Hay dos aspectos importantes:

- Que la especificación de requerimientos sea correcta, en el sentido de las necesidades reales del cliente con todos sus matices.

- Que la especificación de requerimientos sea completa, y no haya ninguna carencia en cuanto a las expectativas del cliente. Añadir a última hora prestaciones que no se habían considerado inicialmente en un sistema de tiempo real puede obligar a rehacer todo el proyecto.

Una de las ventajas del uso de prototipos es que el cliente puede experimentar situaciones que con anterioridad sólo había entendido vagamente.

Para que el prototipo sea viable, debe tener un coste mucho menor que el producto final. Esto solo se puede conseguir con la utilización de los lenguajes de alto nivel, que permiten omitir las consideraciones de detalle, o hasta incluso las fuertes restricciones que impone el tiempo real., También debe poder mostrar un sistema que, aparentemente, sea similar desde el punto de vista funcional al producto que se está desarrollando.

Para reducir el costo efectivo del simulador, se puede desarrollar de forma que parte del diseño sea aprovechable, al menos en parte, para el posterior diseño.